# The Visitor Pattern

●●●

Quang, Patrick, Tobin

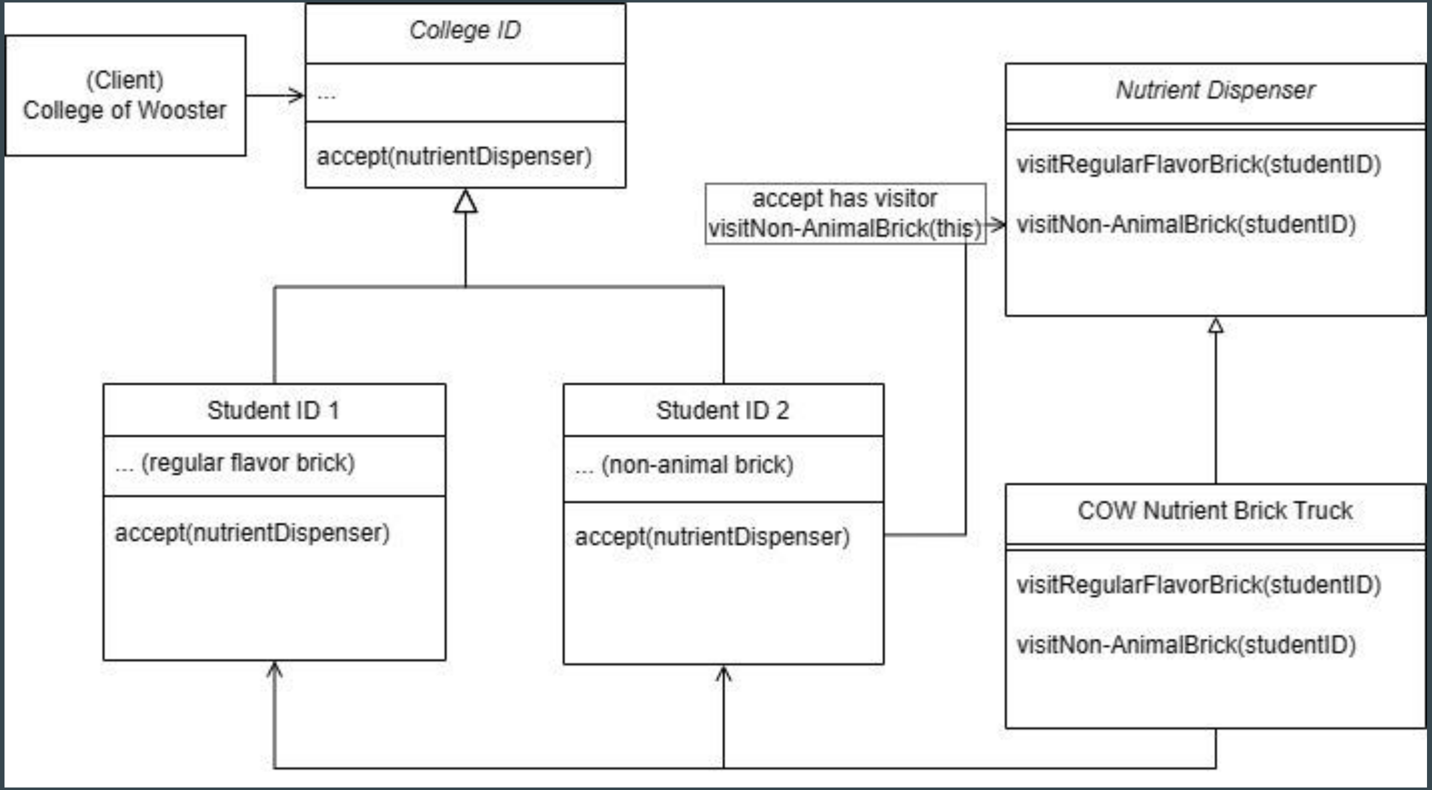
## Real World Example

The year is 2088, and the College of Wooster  is seeking to add dining services after they have been entirely removed in the 2060s after the failed student takeover and subsequent incineration of Lowry Center.

The College functions perfectly without a food delivery system, but seeks to add one on. They don't want to change any of their buildings to add in this service, so they contract with a corporation to bring a food provisionary cart to campus.

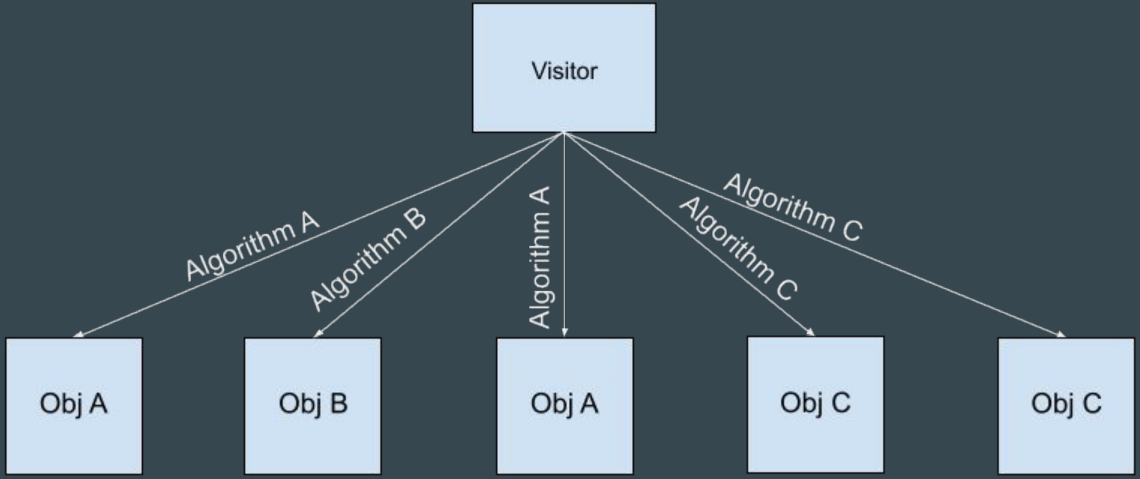
Creatine Dining Services is contracted to serve nutritional ‘composite meal slabs’ to students from a mobile station, but need to know what dietary restrictions students have – regular nutritional brick, non-animal brick, non-gluten brick, or non-animal-non-gluten brick. Because they cannot know the nutrients that each student requires, they wagon will accept the student's ID card, and the mobile wagon will dispense the proper nutrients, taking payment from their account.

# UML Diagram (NuTruck)


(Client)
College of Wooster
College ID
...
accept(nutrientDispenser)
Nutrient Dispenser
visitRegularFlavorBrick(studentID)
visitNon-AnimalBrick(studentID)
accept has visitor
visitNon-AnimalBrick(this)
Student ID 1
... (regular flavor brick)
accept(nutrientDispenser)
Student ID 2
... (non-animal brick)
accept(nutrientDispenser)
COW Nutrient Brick Truck
visitRegularFlavorBrick(studentID)
visitNon-AnimalBrick(studentID)

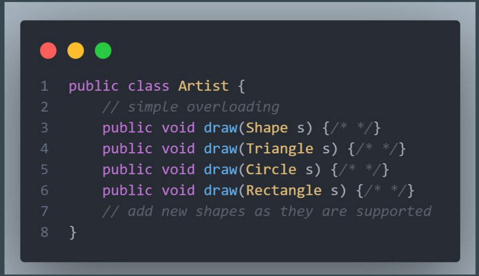
## The Problem

- We need to run an operation through a list of subclasses
- Based on the type of the subclass, the operation you run on it will change accordingly

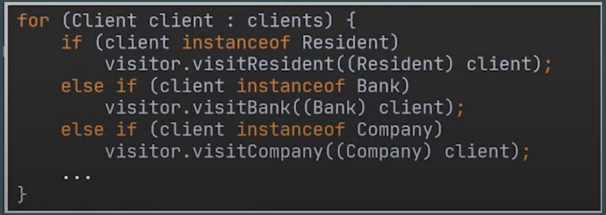


## Solutions

- Using overriding with polymorphism
  - Create a base method in the parent class which all subclasses inherit
  - Have each subclass override this base method and have the iterator call this method
- Problems using overriding
  - Difficult to make future adjustments (must check/make changes to each class file individually)
  - Each class now has more than one role (more functionality to break)
  - Single responsibility (Food bricks  class doesn't know how to make themselves)



```java
public class Artist {
    // simple overloading
    public void draw(Shape s) {/* */}
    public void draw(Triangle s) {/* */}
    public void draw(Circle s) {/* */}
    public void draw(Rectangle s) {/* */}
    // add new shapes as they are supported
}
```



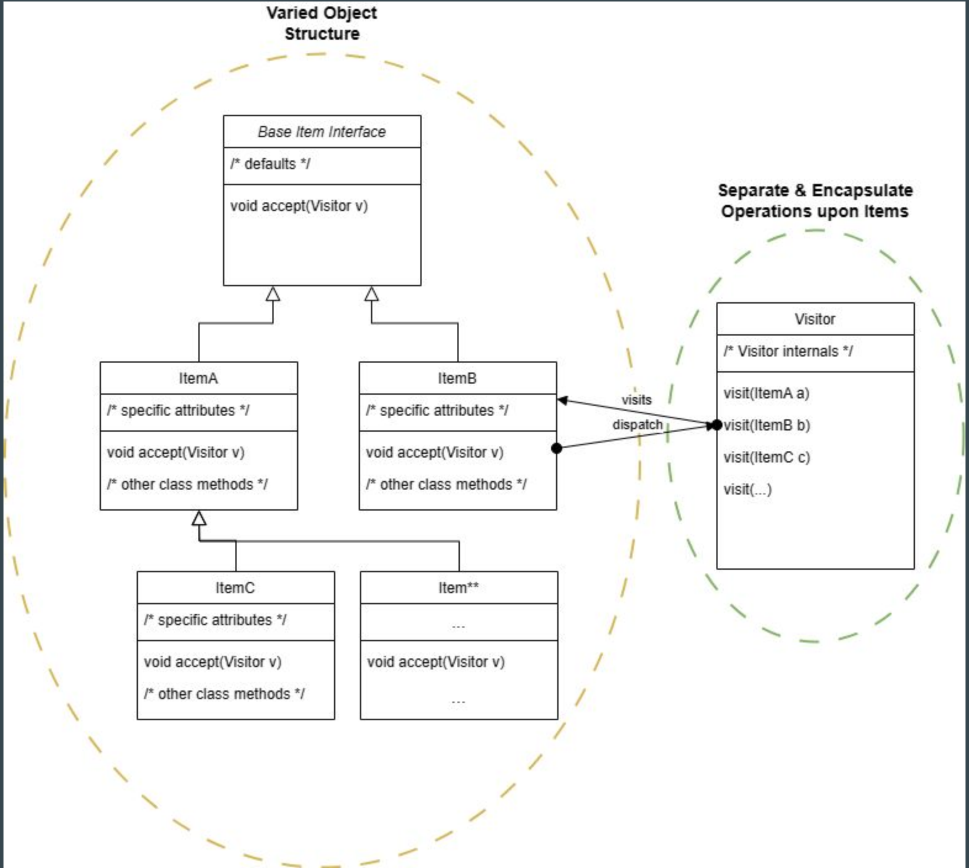
```java
for (Client client : clients) {
    if (client instanceof Resident)
        visitor.visitResident((Resident) client);
    else if (client instanceof Bank)
        visitor.visitBank((Bank) client);
    else if (client instanceof Company)
        visitor.visitCompany((Company) client);
    ...
}
```

- Extract the methods to separate "Visitor" class
  - Now how do we run it?
    - Checking conditionals/method overloading
    - **Double Dispatching**

## What is it?

- Like what we did with M-V-C, but instead for different parts of the logic, separating different functions of the logic from the object
- Separate unrelated methods of an object to a different class
  - Easier management of “unrelated methods”
  - Maintain Single Responsibility Principle
- Allow for a frequently occurring method to be applied across varying inheritance structures
  - Different implementation per specific type
- Exposes ability to use double dispatching
  - Compiler/Runtime specifics, allows for class type information to be checked at runtime

# UML Diagram (In General)



## Geometry Code Example

## Discussion

- What kinds of methods would work well to be encapsulated into a Visitor?
- What existing patterns work well with the Visitor?
- How does this pattern compare to the strategy pattern? Is it simply strategy in disguise?